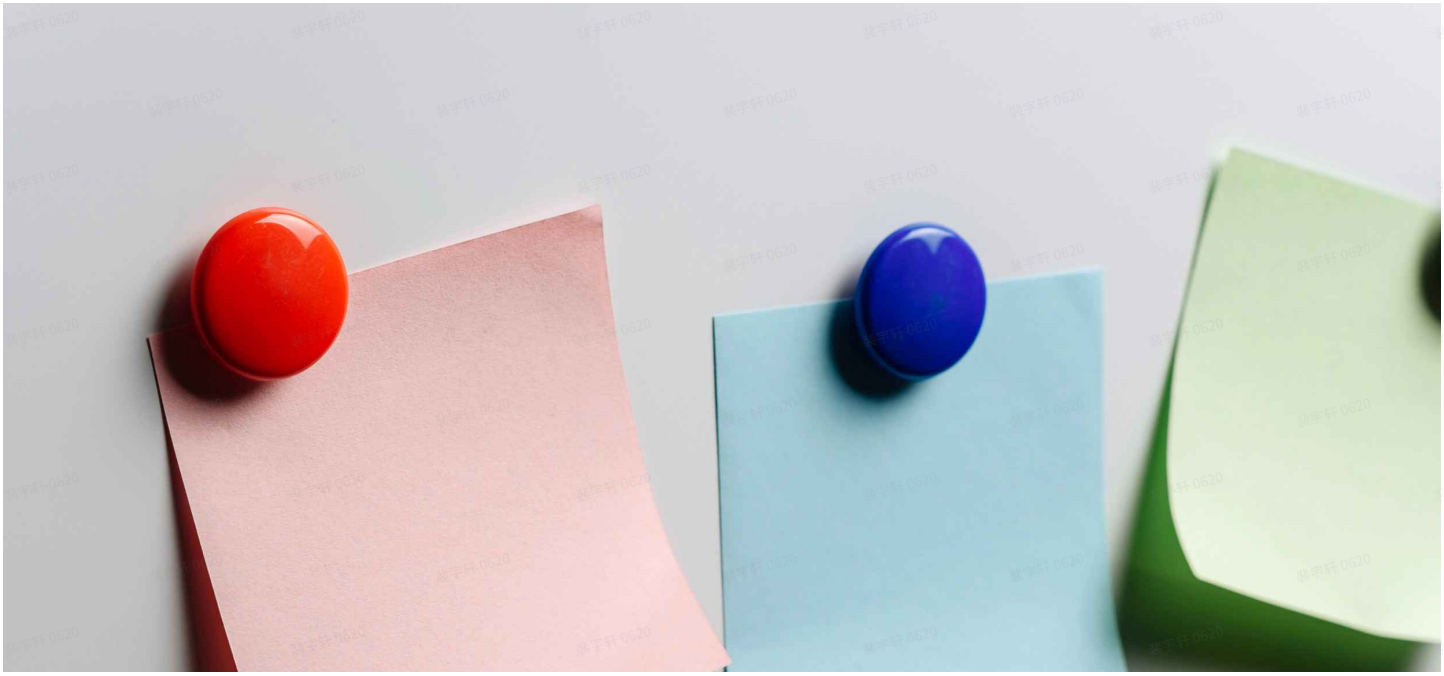


如何优雅的执行任务-并发有序 副本

💭 更多做任务系列

- 如何优雅的执行任务-时序并发
- 如何优雅的执行任务-并发有序 （本文）

TL; DR

💡 前端控制异步任务通常都是比较复杂和苦恼的，代码写起来也不优雅

本文介绍一种多异步任务执行的控制方法，有并发控制、可按序执行（前序任务结束）、可取消任务队列（**中断后续所有任务**）

Why? 一些复杂的交互

任务执行后，中断后续任务

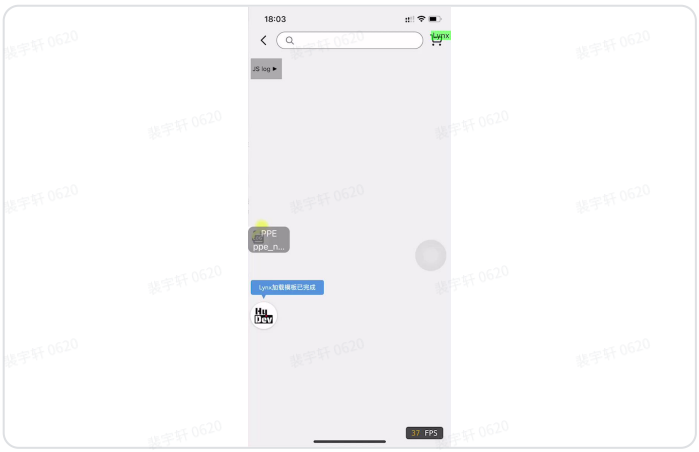
举个例子🌰

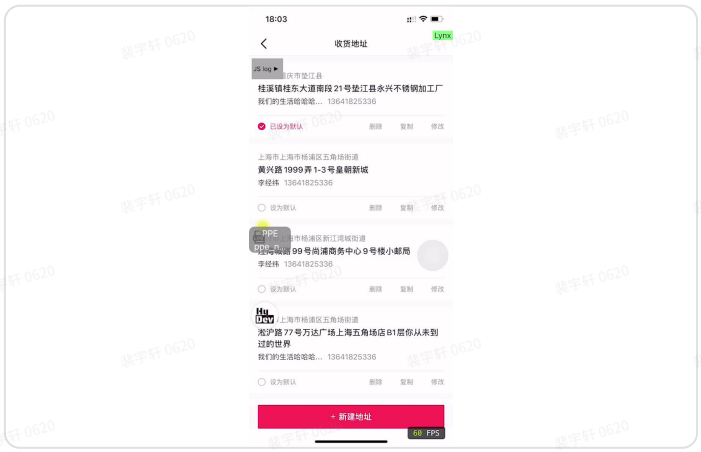
> 地址表单**首屏渲染**后要新增这么个功能 ⬇
>
> - 读取用户剪切板内容，读到内容后 → 调用智能解析 → 展示弹窗内容 → 用户确认填入表单
> - 如果没能成功填入表单，就会继续**执行原来的首屏功能：（case 1）**

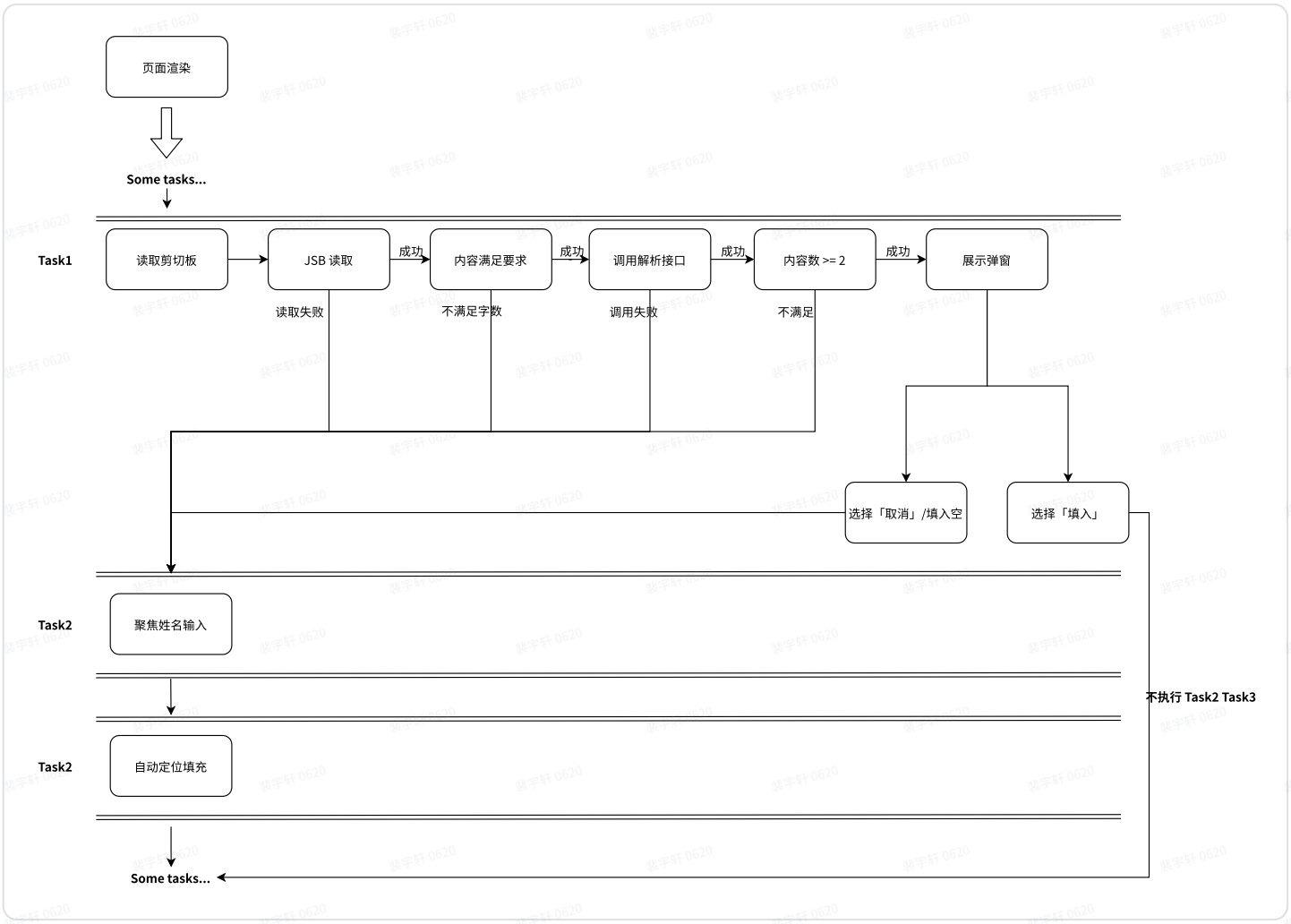
- 自动聚焦输入栏
- 自动定位并填充

- 如果成功填入内容，则**不执行原来的功能（case 2）**

Demo show：**case 1**｜**case 2**







任务抽象

Async Task1：读取剪切板内容 -> 解析接口 -> 弹窗展示等待 callback

- 展示弹窗，**返回【Success】**

    - 在弹窗的 handler，完成「填入」，**中断后续的所有任务，返回【Success】**
    - 未完成「填入」，**继续执行后续 task，返回【Failed】**
- 解析无内容/失败，**返回【Failed】**

Async Task2：异步 focus input

Async Task3：异步调用 JSB -> API -> ...

这些 task 都是“串行”执行

我们一般会怎么写？

```
// componentDidMount
const task1Success = await Task1();
if (!task1Success) {
    Task2();
    Task3();
}

// 弹窗 handler
if (填入) {
    // 无操作
} else if (取消) {
    // 继续执行
    Task2();
    Task3();
}
```

Or 用消息/事件

```
// componentDidMount
Task1(); // 执行失败 or 成功之后 发送 event
subscribe({
    event: 'Task1Finish',
    callback: (res) => {
        if (!res.task1Success) {
            Task2();
            Task3();
        }
    }
})

```

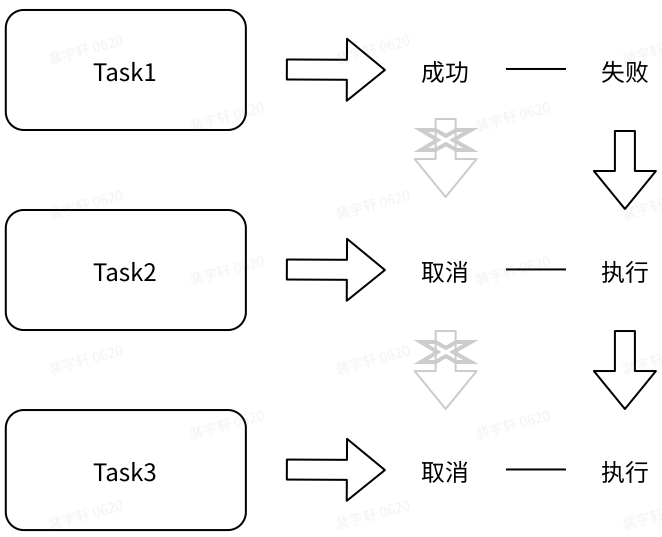
```
13
14 // Task1
15 {
16     ...
17     if (无解析内容) {
18         publish('Task1Finish', { task1Success: false });
19     } else {
20         publish('Task1Finish', { task1Success: true });
21     }
22 }
23
24 // 弹窗 handler
25 if (填入) {
26     // 无操作
27     publish('Task1Finish', { task1Success: false });
28 } else if (取消) {
29     // 继续执行
30     publish('Task1Finish', { task1Success: true });
31 }
```

有什么痛点：

1. Task1 被割裂成两个过程：数据获取，弹窗交互的 handler
2. Task2，Task3 多处调用（非常零散）——如果后续 task 有改动，可能会遗漏 出 bug

流程抽象

> 进一步从执行的角度去看



```
// 伪代码
TaskRunner(Task1); // TaskRunner 能识别到 Task1 完成后是否继续执行后续任务
TaskRunner(Task2); // 也许已经取消了，Task2 不会被执行
TaskRunner(Task3);
// 无需其他场景的二次调用
```

任务调度器（异步）

> 可以 copy 代码在 node 环境执行试试～

任务执行队列

> 有序执行的异步任务 => 排队做任务（核心：利用 await Promise 来实现串行）

```
// 控制任务阻塞的队列
const queue: ((value?: unknown) => void)[] = [];
let executing = false;

// 任务调度器
const execSingleTask = async (task: () => Promise<unknown>) => {
  if (executing) {
    // 当前任务在执行 阻塞 异步等待 把 resolve 加入队列排队 => 排队
    await new Promise(resolve => queue.push(resolve));
  }
  // 更新状态
  executing = !queue.length;
  // 执行任务 => 排队轮到了!
  await task();
  // 消费 resolve 推进任务排队进度
  queue.length && queue.shift()?.();  // => 拿到 resolve 并且 resolve()
  // 更新状态 如果没有任务
  executing = !!queue.length;
};

```

使用 demo

```
// 构造一些异步任务
const timeout =
  (t: number) =>
  <T>(value: T) =>
    new Promise<T>(r => {
      setTimeout(() => {
```

```
        console.log(`${value}`);
        r(value);
      }, t);
    });

const taskA = () => timeout(1000)('taskA');
const taskB = () => timeout(2000)('taskB');
const taskC = () => timeout(1000)('taskC');

// 执行任务队列
console.log('start ----> ');

execSingleTask(taskA);  // 1s 后输出 taskA
execSingleTask(taskB);  // taskA 后 2s 输出 taskB
setTimeout(() => {
  console.log('after 4s');
  execSingleTask(taskC);  // taskB 后 2s 输出 taskC
}, 4000);

```

可取消机制

> 中断后续的所有任务 => 排到你的时候正好都买完啦！

```
execSingleTask(taskA);  // TaskA 执行完 继续
execSingleTask(taskB);  // TaskB 执行完 任务终止
execSingleTask(taskC);  // TaskC 不执行
```

如何实现：任务是按序处理的，当执行到某一任务的时候，检查当前的队列状态是否已经取消

- if 取消：任务不执行，任务状态 reject
- if 不取消：任务继续执行

```
const queue: ((value?: unknown) => void)[] = [];
let executing = false;

// 取消状态
let canceled = false;

// 排队状态更新 抽离
const updateQueue = () => {
  // 消费 resolve 推进排队进度
  queue.length && queue.shift()?.();
```

```
  // 更新状态 如果没有任务
  executing = !!queue.length;
};

const execSingleTask = async (task: () => Promise<unknown>) => {
  if (executing) {
    // 当前任务在执行 阻塞 异步等待 把 resolve 加入队列排队
    await new Promise(resolve => queue.push(resolve));
  }
  // 更新状态
  executing = !queue.length;
  // 执行任务 首先判断当前队列状态
  if (canceled) {
    updateQueue();
    return Promise.reject();
  }

  const res = await task();
  // 如果返回的结果是取消
  if (res === '__canceled') {
    canceled = true;
  }
  updateQueue();
};
```

写个 test demo

```
const taskA = async () => {
  await timeout(1000)('taskA');
  return '__canceled'; // 执行完取消后续任务
};
const taskB = async () => {
  await timeout(2000)('taskB');
  return;
};

const taskC = async () => {
  await timeout(1000)('taskC');
  return;
};

console.log('start ---->');

execSingleTask(taskA).catch(r => console.log('task A canceled'));
execSingleTask(taskB).catch(r => console.log('task B canceled'));
```

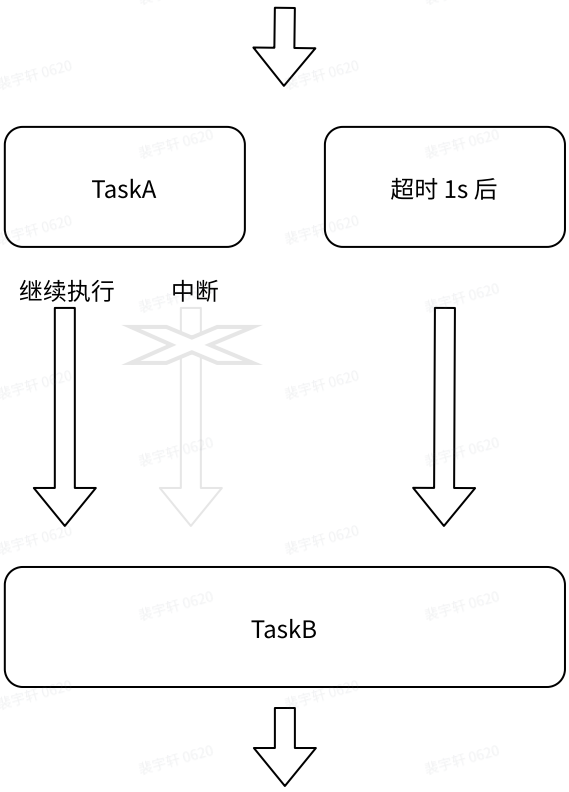
```
setTimeout(() => {
  console.log('after 4s');
  execSingleTask(taskC).catch(r => console.log('task C canceled'));
}, 4000);
```

输出结果

```
start ---->
1000_taskA
task B canceled
after 4s
task C canceled
```

到这里，我们就完成了一个可以中断的任务调度器，来实现我们想要的效果！

*但是呢，另一个需求又有一个比较复杂的交互：在需要中断任务的前提下，加入了**超时机制***



可以把超时抽象一个任务——TaskTimeout

TaskA 和 TaskTimeout 两个人一起排队，TaskA 执行完毕后，可以控制后续的任务是否中断/继续，如果 TaskTimeout 先做完了（即 TaskA 超时了）就可以继续执行剩下的任务。

一起排队？——加入并发控制！

并发控制

实现思路：其实只需要多一个任务数量的判断，来控制是否排队，直接看最终版的代码！

```ts
export enum TaskResult {
  CANCEL = '__cancel__'
}

export type TaskWithCancel = () => void | TaskResult | Promise<TaskResult | void>;

/**
 * 可控制并发数 且可中断的 任务调度器 @lijingwei.xyz
 * @param limit 最多执行的 task 数量
 * @returns
 */
export const runTaskWithLimitAndStop = (limit: number) => {
  let count = 0;
  let canceled = false; // 如果取消了 下一个 push 进来的任务就直接 reject 了
  const blockQueue: ((value?: unknown) => void)[] = [];

  const pass = () => {
    count--;
    blockQueue.length && blockQueue.shift()?.(); // resolve 放行
  };

  return async (fn: TaskWithCancel) => {
    count++;
    if (count > limit) {
      // 用 await resolve 来阻塞 线程执行 等上一个 resolve 之后再开始后续的 fn()
      // 直到下面的 fn 执行完了 blockQueue 的 resolve 了 才会继续 fn()
      await new Promise(resolve => blockQueue.push(resolve));
    }
    try {
      if (canceled) {
        pass();
        return Promise.reject(TaskResult.CANCEL);
      }
      const result = await fn();
      if (result === TaskResult.CANCEL) {
        // 中断
        canceled = true;
      }
      pass();
      return result;
    } catch (e) {
```

```
      pass();
      return Promise.reject(e);
    }
  };
};

```

问题解决

> 代码 demo（reactlynx）

首屏剪切板逻辑

```
const singleTaskRunner = runTaskWithLimitAndStop(1);

//-------- run task 没有 await
// 看是否需要读取剪切板
if (initCheckClipBoard) {
  // 剪切板读取 & 填入成功 会取消下面两个任务的执行
  // 会在弹窗的 handler 里面 完成继续 or 中断 的操作
  singleTaskRunner(this.readClipBoardTask).catch(() => {});
}
singleTaskRunner(this.autoFillLocationTask).catch(() => {}); // catch 可能是 cancel 了
singleTaskRunner(this.autoFocusNameTask).catch(() => {}); // catch 可能是 cancel 了
//-------- task end
```

超时并发控制

```
// 最大并发两个异步任务 超时 and 预咨询
const taskRunner = runTaskWithLimitAndStop(2);

// 获取挽留预咨询 + 超时 超时后 fallbcak 到原逻辑
if (this.needQueryApiWhenLeave) {
  // 判断是否需要 预咨询 挽留渲染内容
  const timer = new Timer(this.detainmentRenderTimeoutMs);
  // 构造 task
  const renderIncentiveTask: TaskWithCancel = async () => {…
  };
  // 构造 task
  const timeoutTask: TaskWithCancel = async () => {…
  };

  // 2 个任务并行执行 超时任务 and 查券任务
  taskRunner(renderIncentiveTask);
  taskRunner(timeoutTask);
}
// 这里需要 await 直到这个节点 结束 才 show
// 兜底逻辑 查询优惠信息
await taskRunner(checkDiscountTask).catch(() => {}); // catch 了 可能是被 cancel 了

// 挽留弹窗之后 加这个通用参数 是否展示了优惠/券信息
```

好处 & 待改进的

好处：

- 使用简单，代码可读性高，不割裂，不用到处去找其他调用的地方，不会遗漏
- 扩展性还不错，任务的控制状态可以自由改动

待改进：

- 任务一旦取消，无法重新启动，需要重新构造队列
- 有一定的理解成本...